

Analizador Léxico y

sintáctico

### -----------------

Álvarez González Ian

García Oviedo Jaasiel Osmar

López López Ulysses

Domínguez Cisneros Alexis Saul

Gpo 1.

Compiladores |

2020-1

Analizador léxico

Análisis del problema:

Se debe elaborar un programa capaz de interpretar las 27 gramáticas dejadas por el profesor, para esto debe crear un analizador Léxico.

Diseño de solución:

Para poder hacer este analizador decidimos usar el lenguaje lex o flex, que, precisamente se basa en el uso de expresiones regulares para la identificación ya sea de patrones o cadenas de lenguaje, lo cual nos ayudará con las cadenas que le queramos introducir. Ahora bien, a continuación, se mostrarán algunos de los diferentes componentes léxicos o tokens que deberá reconocer nuestro analizador léxico, así como sus respectivas expresiones regulares, las cuales hemos visto durante el transcurso del semestre:

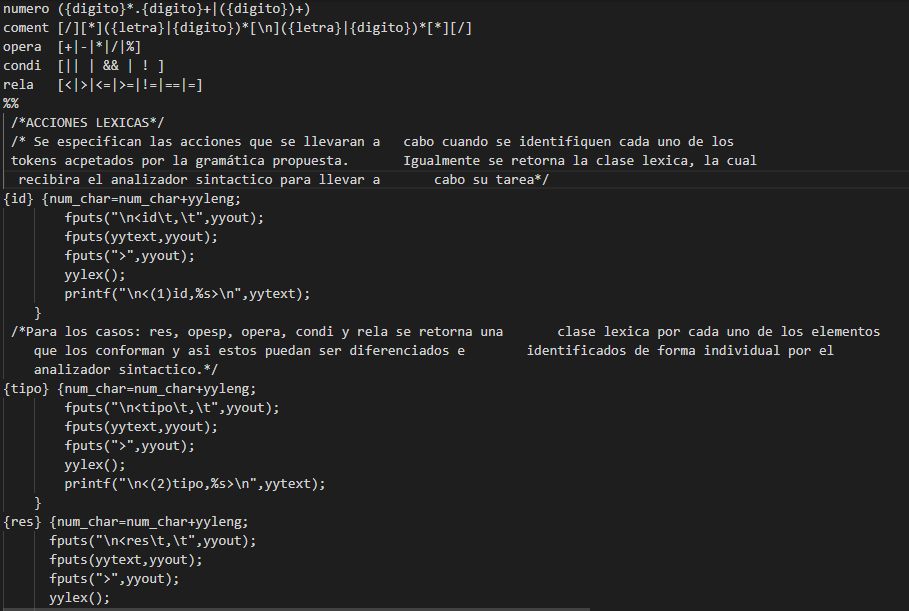
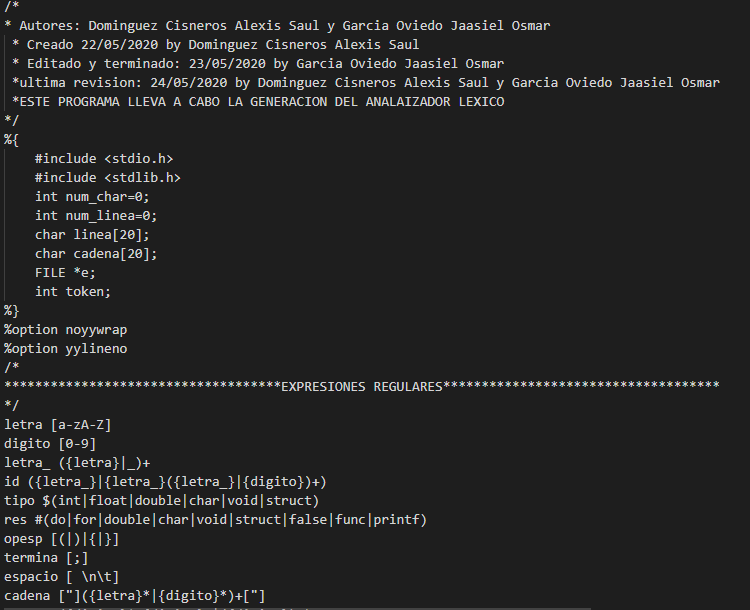
|  |  |  |
| --- | --- | --- |
| Clase o id | Nombre | Expresión regular |
| 0 | Reservadas | (do|for|double|char|void|struct|false|func|printf) |
| 1 | digito | [0-9] |
| 2 | Reales | [Ee][+-]?[0-9]{1,2} |
| 3 | Exponentes | (({enteroNum}\.[0-9]\*|\.{enteroNum}){exponente}?|{enteroNum}{exponente}) |
| 4 | Letra | [a-zA-z] |
| 4 | Letra\_ | ({letra}|\_)+ |
| 5 | Id | ({letra\_}|{letra\_}({letra\_}|{digito})+) |
| 6 | Operador | [+|-|\*|/|%] |
| 7 | Cadena | ["]({letra}\*|{digito}\*)+["] |
| 8 | Cáracter | ' . ' |
| 9 | OpeEsp | [(|)|{|}] |
| 10 | Condicional | [|| | && | ! ] |
| 11 | Comentario | [/][\*]({letra}|{entero})\*[\n]({letra}|{entero})\*[\*][/] |
| 12 | Relacional | [<|>|<=|>=|!=|==|=] |
| 13 | Espacio | [ \n\ t ] + |

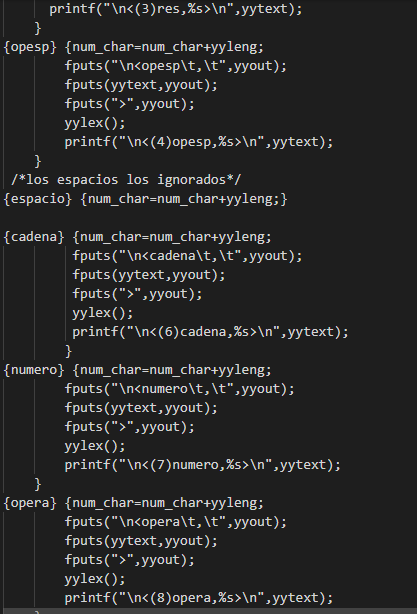
A continuación, mostraremos las gramáticas que nos dieron:

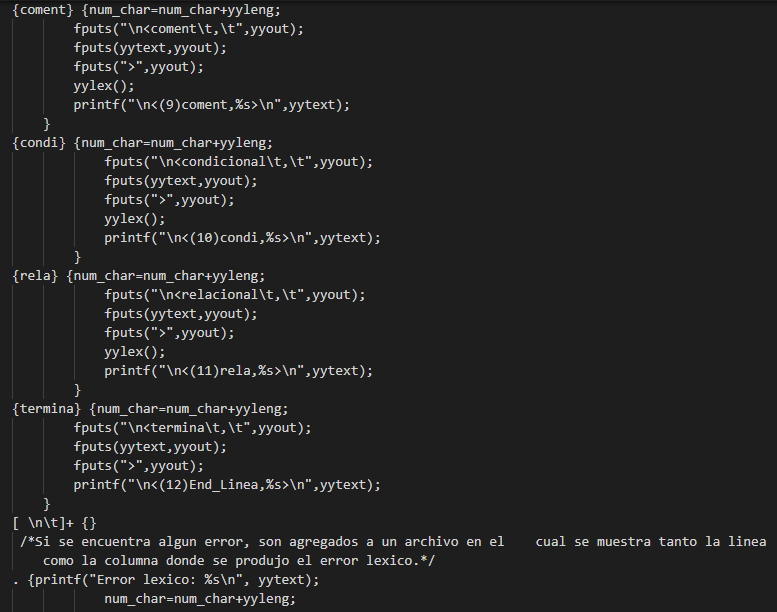
Gramatica

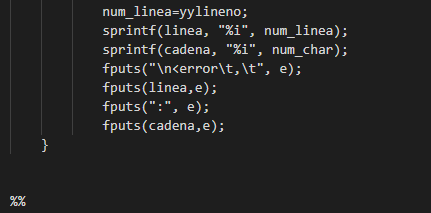
sin: significa sin tipo, car: tipo carácter

|  |  |
| --- | --- |
| Gramaticas | |
| 1. programa → declaraciones funciones | 2. declaraciones → tipo lista\_var; declaraciones| tipo registro lista\_var; declaraciones| ε |
| 3. tipo\_registro → estructura inicio declaraciones fin | **4. tipo → base tipo\_arreglo** |
| 5. base → ent | real | dreal | car | sin | **6. tipo\_arreglo → [num] tipo arreglo | ε** |
| 7. lista var → lista var, id | id | **8. funciones → def tipo id(argumentos) inicio declaraciones sentencias fin funciones| ε** |
| 9. argumentos → listar\_arg | sin | **10. lista\_arg → lista\_arg, arg | arg** |
| 11. arg → tipo\_arg id | **12. tipo\_arg → base param\_arr** |
| 13. param\_arr → [ ] param\_arr | ε | **14. sentencias → sentencias sentencia | sentencia** |
| 15. sentencia → si e\_bool entonces sentencia fin | si e\_bool entonces sentencia sino sentencia fin | mientras e\_bool hacer sentencia fin | hacer sentencia mientras e\_bool; | según (variable) hacer casos predeterminado fin | variable := expresion ;  | escribir expresion ;  | leer variable ; | devolver;  | devolver expresion;  | terminar;  | inicio sentencias fin | **16. casos → caso num: sentencia casos | caso num: sentencia** |
| 17. predeterminado → pred: sentencia | ε | **18. e\_bool → e\_bool o e\_bool | e\_bool y e\_bool | no e\_bool | ( e\_bool )| relacional | verdadero | falso** |
| 19. relacional → relacional oprel relacional | expresion | **20. oprel → > | < | >= | <= | <> | =** |
| 21. expresion → expresion oparit expresión  | expresion % expresion | (expresion ) | id  | variable | num | cadena | caracter | id(parametros) | **22. oparit → + | − | ∗ | /** |
| 23. variable → dato est\_sim | arreglo | **24. dato est\_sim → dato est\_sim .id | id** |
| 25. arreglo → id[expresion] | arreglo[expresion] | **26. parametros → lista\_param | ε** |
| 27. lista\_param → lista\_param, expresion | expresion |  |

Implementación: 

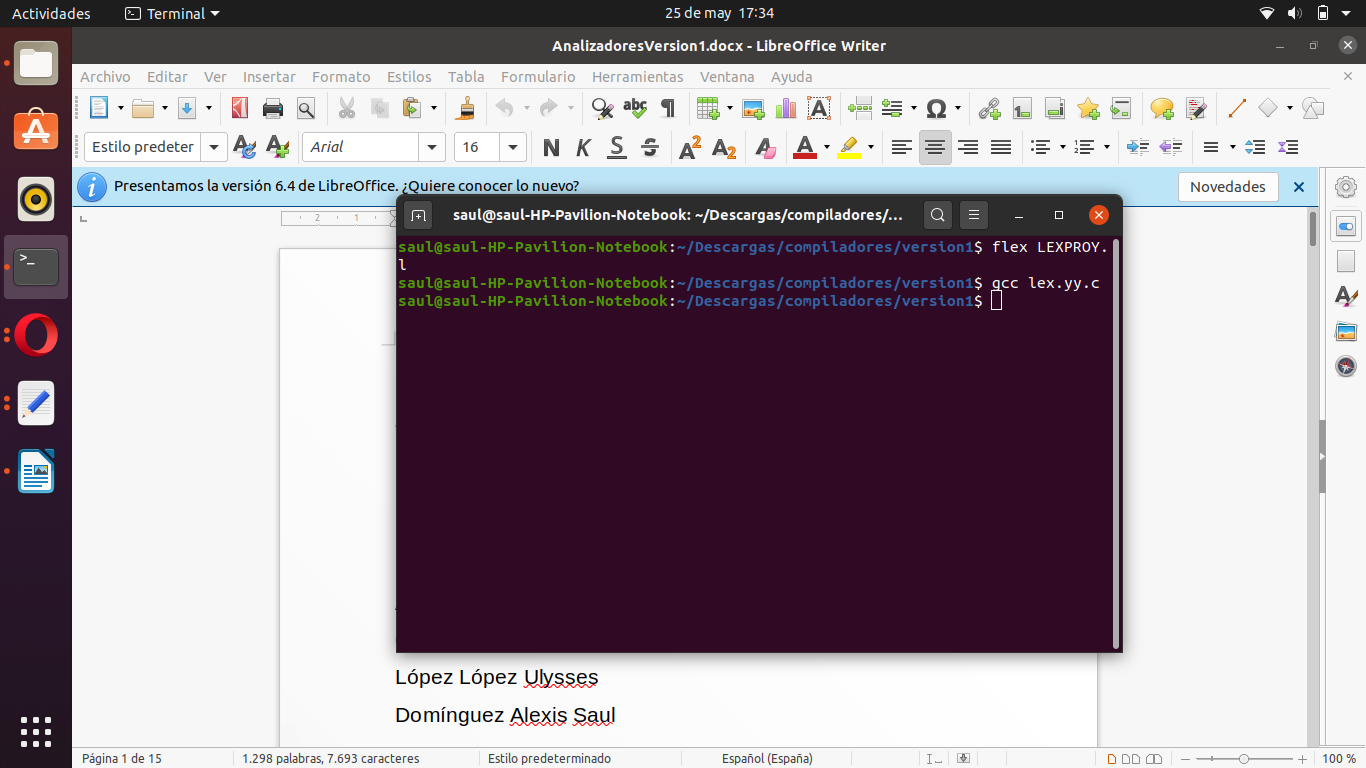






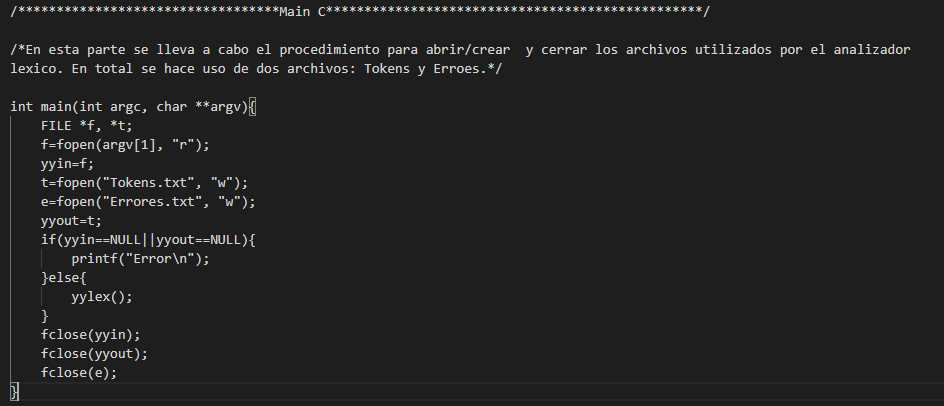
Compilar

1. Para poder ejecutar el programa tenemos que ir a la consola de nuestro equipo.
2. Tenemos que ingresar al fichero donde se encuentra nuestro archivo.
3. Después ingresamos el siguiente comando: *flex “nombre.l”,* si no genera ningún error podemos proceder con el siguiente paso.
4. Ejecutamos el siguiente comando: *gcc lex.yy.c*
5. Se muestra el resultado.



Compilo correctamente

Código main.c



Análisis Sintáctico

Análisis del problema:

El problema principal es poder obtener una correcta interpretación de una serie de gramáticas propuestas por el profesor, esto forma parte de un conjunto de problemas, los cuales tienen como objetivo un proyecto final el cual es construir un compilador.

En el análisis sintáctico se deben tomar varias decisiones, la gramática en algunas partes presenta ambigüedad, sin embargo; esta se puede ignorar utilizando en bison la precedencia de operadores y dejar que el programa la resuelva de manera automática o eliminar la ambigüedad antes de transcribir la gramática en bison.

Otra consideración que se debe tener en cuenta para resolver la ambigüedad del if - else, es que se debe asignar una precedencia a else como si fuera el operador de mayor precedencia. Ahora bien, teniendo en cuenta las gramáticas que se nos dieron, hicimos sus producciones, así como sus respectivas reglas semánticas como se mostrarán a continuación:

|  |  |
| --- | --- |
| Producción | Regla semántica |
| P→ DF | F.tipo=D.tipo |
| D→ TL; D | L.TIPO=T.TIPO  L.DIM=T.DIM |
| D→ Ɛ | D.tipo=D.base |
| T →  int | T.TIPO=INT  T.DIM=2 |
| T→ float | T.TIPO=FLOAT  T.DIM=8 |
| T→ double | T.TIPO=DOUBLE  T.DIM=16 |
| T→ char | T.TIPO=CHAR  T.DIM=1 |
| T→ void | T.TIPO=VOID  T.DIM=0 |
| L → L1, id A | L1.TIPO=L.TIPO  IF SYMTAB.ADD(ID,L.TIPO,DIR,’VAR’,-) != -1  DIR=DIR+TYPE.TAB.GETDIM(L.TIPO)  ELSE  ERROR  A.BASE=L1.TIPO |
| L→ id A | SYMTAB.ADD(ID,LEXVAL,L.TIPO)  A.BASE=L.TIPO |
| A → [ num ] A1 | IF NUM.TIPO=INT || NUM.TIPO=FLOAT || NUM.TIPO=DOUBLE || NUM.TIPO=CHAR || NUM.TIPO=VOID  A1.BASE=A.BASE  A.TIPO=TYPETAB.ADD(“ARRAY”, NUM.VAL,A1.TIPO”)  ELSE  ERROR |
| A →Ɛ | A.TIPO=A.BASE |
| F → func T id ( ARG ) { D S} F |  |
| F →Ɛ | F.TIPO=F.BASE |
| ARG → L\_A | L\_A.SIG=NUEVAETQ()  ARG.CODIGO=L\_A.CODIGO||ETQ(L\_A.SIG) |
| ARG →Ɛ | ARG.TIPO=ARG.BASE |
| L\_A → L\_A , T id P\_A | L\_A.TIPO=T.TIPO  IF SYMTAB.ADD(ID,L\_A.TIPO,DIR,’VAR’,-) != -1  DIR=DIR+TYPE.TAB.GETDIM(L\_A.TIPO)  ELSE  ERROR  P\_A.TIPO=T.TIPO |
| L\_A →T id P\_A | IF SYMTAB.ADD(ID,L\_A.TIPO,DIR,’VAR’,-) != -1  DIR=DIR+TYPE.TAB.GETDIM(L\_A.TIPO)  ELSE  ERROR  P\_A.TIPO=T.TIPO |
| P\_A → [ ] P\_A | P\_A.TIPO=L\_A.TIPO |
| P\_A →Ɛ | P\_A.TIPO=P\_A.BASE |
| C →  C1 || C2 | C1.VERDADERO=C.VERDADERO C1.FALSO=NUEVAETIQUETA() C2.VERDADERO=C.VERDADERO C2.FALSO=C.FALSO C.CODIGO=C1.CODIGO||ETQ(C1.FALSO)||C2.CODIGO |
| C→ C1&&C2 | C1.VERDADERO=NUEVAETIQUETA()C1.FALSO=C.FALSO C2.VERDADERO=C.VERDADERO C2.FALSO=C.FALSO C.CODIGO=C1.CODIGO||ETQ(C1.VERDADERO)||C2.CODIGO |
| C→ !C1 | C1.VERDADERO=C.FALSO  C1.FALSO=C.VERDADERO  C.CODIGO=C1.CODIGO |
| C→  ( C1 ) | C.CODIGO=C1.CODIGO |
| C→ E1 R E2 | C.CODIGO=IF E1.DIR R.VAL E2.DIR  GOTO C.VERDAERO  GOTO C.FALSO |
| C→ TRUE | C.DIR=RES.LEXVAL |
| C→ FALSE | C.DIR=RES.LEXVAL |
| E→ E1+E2 | E.VAL=E1.VAL+E2.VAL |
| E→ E1-E2 | E.VAL=E1.VAL-E2.VAL |
| E→ E1\*E2 | E.VAL=E1.VAL\*E2.VAL |
| E→ E1/E2 | E.VAL=E1.VAL/E2.VAL |
| E→ E1%E2 | E.VAL=E1.VAL%E2.VAL |
| expresión  :expresion MAS expresión  {  E  } | |
| E→V\_A | E.VAL=V\_A.VAL |
| E→ cadena | E.DIR=cadena.lexval |
| E→numero | E.DIR=numero.lexval |
| E→caracter | E.DIR=caracter.lexval |
| E→ id(P\_A) |  |
| V\_A→id[E] |  |
| V\_A→V\_A[E] |  |
| P\_A→ Ɛ | P\_A.TIPO=P\_A.BASE |
| P\_A→ L\_P | P\_A.CODIGO=L\_P.CODIGO |
| L\_P→E | L\_P.VAL=E.VAL |
| L\_P→L\_P,E |  |

De las anteriores reglas las que utilizaremos en el programa son:

|  |  |
| --- | --- |
| 1 | P → DF |
| 2 | D → T L;D | Ɛ |
| 3 | T → int | float | double | char | void | struct {D} |
| 4 | L → L, id A | id A |
| 5 | A → [ num ] A | Ɛ |
| 6 | F → func T id ( ARG ) { DS } F | Ɛ |
| 7 | ARG → L\_A |  Ɛ |
| 8 | L\_A → L\_A , T id P\_A | T id P\_A |
| 9 | P\_A → [ ] P\_A | Ɛ |
| 10 | S → SS | if ( C ) S | if ( C ) S else S | while ( C ) S | do S while ( C ); | for  ( S ; C ; S ) S | P\_I = E ; | return E; | return; | { S } | switch ( E ) { CAS PRED } | break; | print E; |
| 11 | CAS → case: num S PRED |  Ɛ |
| 12 | PRED → default: S | Ɛ |
| 13 | P\_I → id | V\_A | id.id |
| 14 | V\_A → id [ E ] | V\_A [ E ] |
| 15 | E → E + E | E – E | E \* E | E / E | E %  E | V\_A | cadena | num | carácter | id (PAR) |
| 16 | PAR → Ɛ | L\_P |
| 17 | L\_P → L\_P , E | E |
| 18 | C → C || C | C && C | ! C | ( C )  | E R E | true | false |
| 19 | R → < | > | >= | <= | != | == |

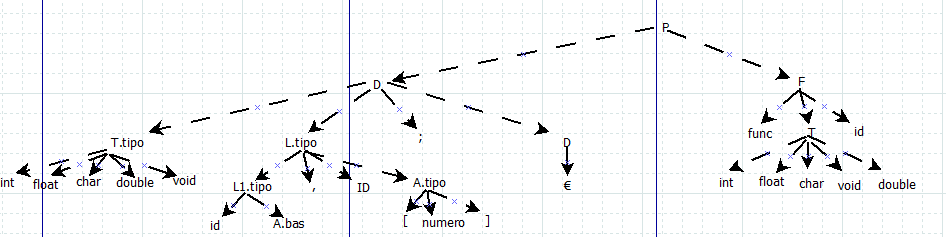
Los siguientes árboles fueron generados para llevar a cabo el análisis sintáctico

Ilustración 1: Árbol 1

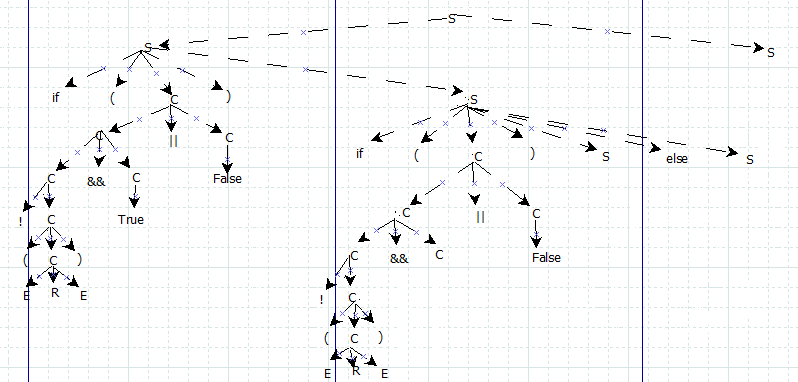


Ilustración 2: Árbol

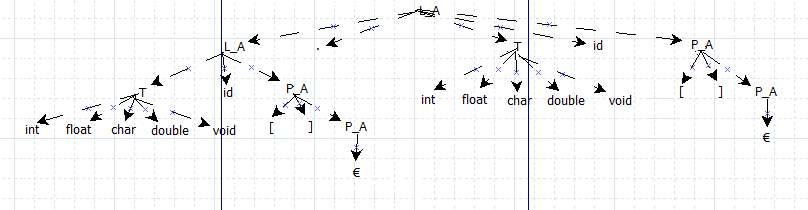


Ilustración 3:Árbol 3

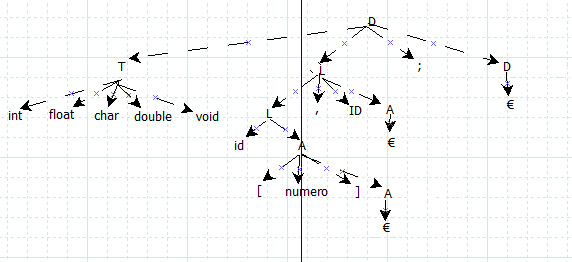
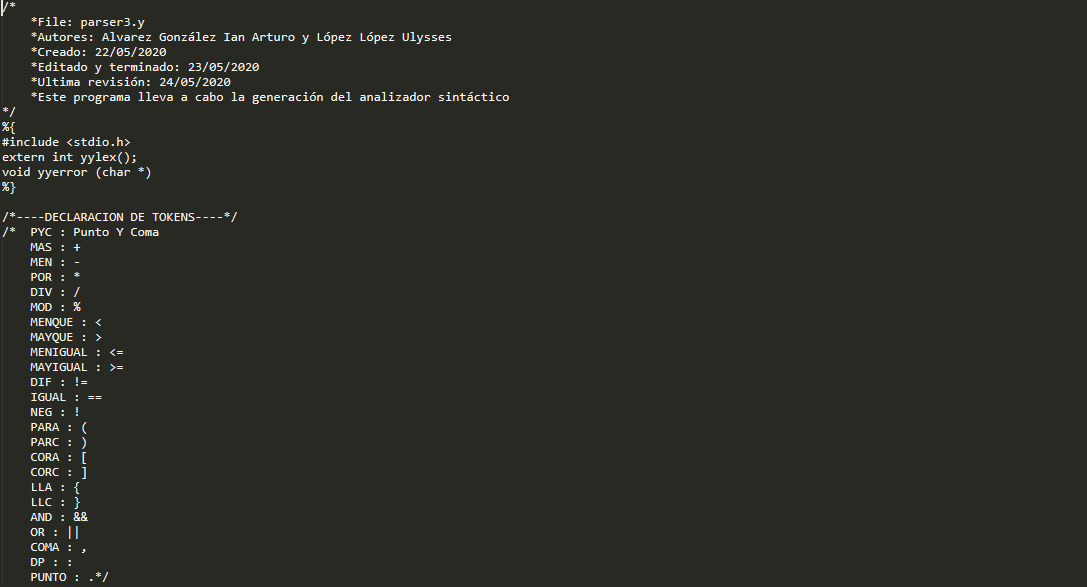
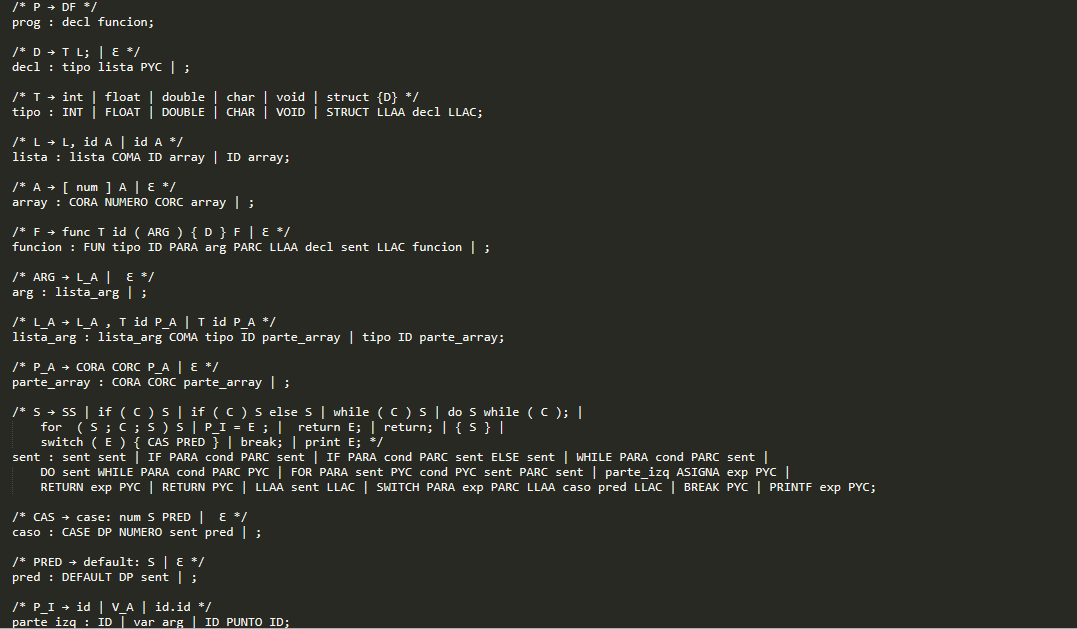
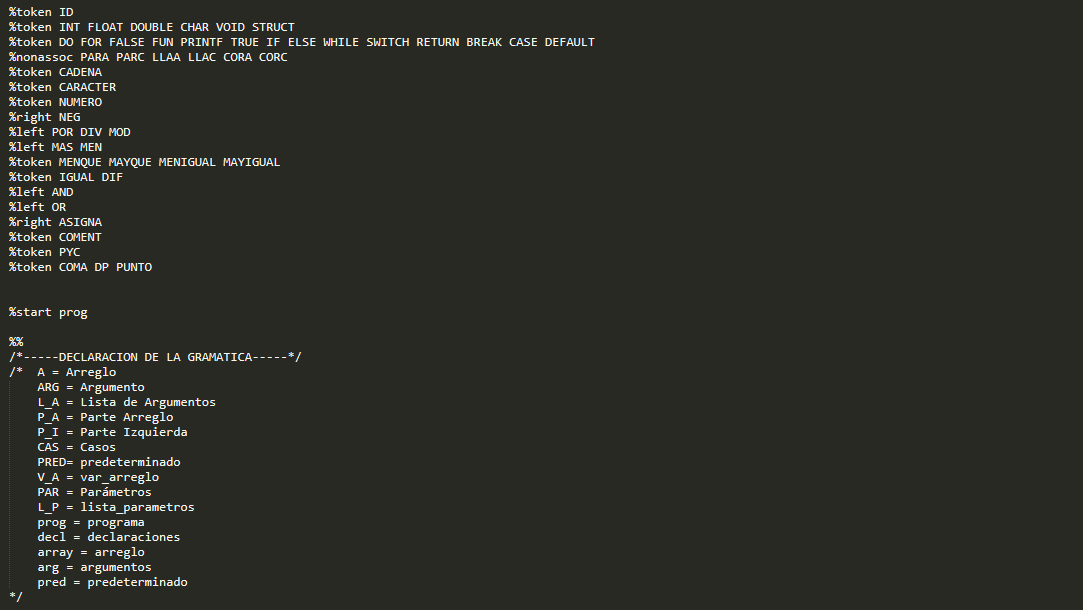
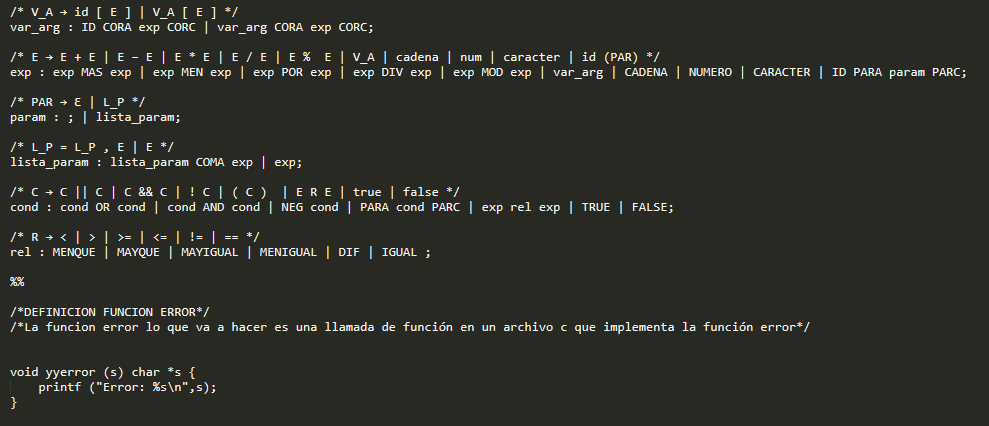


Ilustración 4: Árbol 4

Implementación:







Compilación:

1. Para poder ejecutar el programa tenemos que ir a la consola de nuestro equipo.
2. Tenemos que ingresar al fichero donde se encuentra nuestro archivo.
3. Después ingresamos el siguiente comando: *bison -yd “nombre.y”.*

